EFFICIENT SPARSE MATRIX-MATRIX PRODUCTS USING COLORINGS

MICHAEL MCCOURT*, BARRY SMITH*, AND HONG ZHANG *

Abstract. Sparse matrix-matrix products appear in multigrid solvers and computational methods for graph theory. Some formulations of these products require the inner product of two sparse vectors, which have inefficient use of cache memory. In this paper, we propose a new algorithm for computing sparse matrix-matrix products by exploiting matrix nonzero structure through the process of graph coloring. We prove the validity of this technique in general and demonstrate its viability for multigrid methods used to solve three-dimensional boundary value problems.

1. Introduction. Matrix-matrix multiplication is a fundamental linear algebra operation [4]. The operation of concern in this work is

$$\mathsf{C} = \mathsf{A}\mathsf{B}^T,$$

which arises in algebraic multigrid and certain graph algorithms. This operation is well defined when $\mathsf{A} \in \mathbb{R}^{m \times r}$ and $\mathsf{B} \in \mathbb{R}^{n \times r}$ and will produce $\mathsf{C} \in \mathbb{R}^{m \times n}$. Our results are equally applicable to complex-valued matrices, but we omit that discussion for simplicity.

The multiplication operation is defined such that the jth column of the ith row of C (denoted as $\mathsf{C}(i,j)$) is calculated as

$$\mathsf{C}(i,j) = \sum_{k=1}^{r} \mathsf{A}(i,k)\mathsf{B}^T(k,j).$$

This can also be written as the *inner product* of the ith row of A, denoted as $\mathsf{A}(i,:)$, and the jth column of B^T, denoted as $\mathsf{B}^T(:,j)$:

$$\mathsf{C}(i,j) = \mathsf{A}(i,:)\mathsf{B}^T(:,j). \tag{1.1}$$

Several mechanisms exist for computing the matrix-matrix product, each of which is preferable in certain settings. For instance, C can be constructed all at once with a sum of r outer products (rank-1 matrices),

$$\mathsf{C} = \sum_{k=1}^{r} \mathsf{A}(:,k)\mathsf{B}^T(k,:). \tag{1.2}$$

For an approach between using (1.1) to compute one value at a time and using (1.2) to compute C all at once, we can compute one row or column at a time or compute blocks of C using blocks of A and B^T.

While all these techniques yield the same result, they may not be equally preferable for actual computation because of the different forms in which a matrix may be stored. Many sparse matrix algorithms have memory access outpace floating-point operations as the computational bottleneck [11]; this demands diligence when choosing a storage format so as to minimize memory traffic. In this paper, we begin with conventional matrix storage, namely, compressed sparse row format (CSR) for sparse matrices and Fortran-style column-major array storage for dense matrices [2] and then transform the storage format for higher efficiency.

One approach to computing $\mathsf{A}\mathsf{B}^T$ would be to compute B^T from B and store it as a CSR matrix, but this reorganization requires a significant amount of memory traffic. The availability of the columns of B^T would seem to indicate that the inner product computation (1.1) is the preferred method. Unfortunately, computing the inner product of sparse vectors is an inefficient operation because of the low ratio of flops to cache misses (see Section 1.1). Our goal is to produce a data structure and algorithm which will efficiently compute the sparse matrix-matrix product $\mathsf{C} = \mathsf{A}\mathsf{B}^T$ using inner products.

In the rest of this section we introduce sparse inner products and matrix coloring. In Section 2 we analyze matrix coloring applied to the sparse matrix product $\mathsf{C} = \mathsf{A}\mathsf{B}^T$, which allows us to instead compute C by evaluating the inner product of sparse and dense vectors. In Section 3 we propose algorithms for computing

---

*Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439. Emails: (`mccomic,` `bsmith, hzhang)@mcs.anl.gov`

matrix products with matrix colorings and consider some practical implementation issues. Numerical results are presented in Section 4, which show the benefits of this coloring approach for matrix products appearing in multigrid solvers for three dimensional PDEs from PFLOTRAN [8] and from the PETSc library distribution [2]. We conclude the paper in Section 5 with a brief discussion of future work.

1.1. Sparse inner products. When matrices A and B are stored in compressed sparse row format, computing $\mathsf{C} = \mathsf{A}\mathsf{B}^T$ requires the inner product between sparse vectors; this section discusses the inefficiency (in number of flops per cache miss) of sparse inner products. Algorithm 1 shows how inner products between sparse vectors in $\mathbb{R}^n$ may be computed.

Algorithm 1 Sparse-Sparse Inner Product of $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$; $\boldsymbol{x}, \boldsymbol{y}$ have n_x, n_y nonzeros, respectively.

1: $i = 1$; $j = 1$; $\boldsymbol{x}^T \boldsymbol{y} = 0.0$
2: **while** $(i \leq n_x$ and $j \leq n_y)$ **do**
3:     if (index$(\boldsymbol{x}, i) <$ index$(\boldsymbol{y}, j)$) { $i = i + 1$; }
4:     else if (index$(\boldsymbol{x}, i) >$ index$(\boldsymbol{y}, j)$) { $j = j + 1$; }
5:     else { $\boldsymbol{x}^T \boldsymbol{y} = \boldsymbol{x}^T \boldsymbol{y} + \boldsymbol{x}(\text{index}(\boldsymbol{x}, i))\boldsymbol{y}(\text{index}(\boldsymbol{y}, j))$; $i = i + 1$; $j = j + 1$;}
6: **end while**

Because $\boldsymbol{x}$ and $\boldsymbol{y}$ are stored in compressed form, only the n_x and n_y nonzero values in each vector respectively are stored, along with the rows to which they belong. The function "index" accepts a vector $\boldsymbol{x}$ and an integer $1 \leq i \leq n_x$ and returns the index of the ith nonzero; the ith nonzero is then accessed as $\boldsymbol{x}(\text{index}(\boldsymbol{x}, i))$.

Algorithm 1 iterates through the nonzero entries in each vector until finding the collisions between their indices; each collision is then accumulated to evaluate the inner product. While the number of floating-point operations (flops) for this algorithm is only $O(\text{number of collisions between nonzeros})$, the required memory access is inefficient. The indices of both vectors must be traversed completely, requiring $O(n_x + n_y)$ memory accesses in a while-loop.

This poor ratio of computation to memory access is one motivation behind our work. Contrast Algorithm 1 with the inner product of a sparse and dense vector in Algorithm 2.

Algorithm 2 Sparse-Dense Inner Product of $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$; $\boldsymbol{x}$ has n_x nonzeros and $\boldsymbol{y}$ is dense.

1: $\boldsymbol{x}^T \boldsymbol{y} = 0.0$
2: **for** $(i = 1, \ldots, n_x)$ **do**
3:     $\boldsymbol{x}^T \boldsymbol{y} = \boldsymbol{x}^T \boldsymbol{y} + \boldsymbol{x}(\text{index}(\boldsymbol{x}, i))\boldsymbol{y}(\text{index}(\boldsymbol{x}, i))$
4: **end for**

Algorithm 2 performs $O(n_x)$ flops, because even if some of the values in $\boldsymbol{y}$ are zero, they are all treated as nonzeros. This is greater than the sparse-sparse inner product but with roughly the same level of memory accesses: $O(2n_x)$ in a for-loop. Essentially, a sparse-dense inner product would do more flops per memory access. Our interest is not in this small trade-off but in compressing multiple sparse vectors into a single, dense vector; this will significantly increase the ratio of flops to memory accesses, without (we hope) introducing a harrowing number of zero-valued nonzeros into the computation. If this can be done effectively, then the inefficient sparse-sparse inner products used to compute sparse matrix products can be replaced with a reduced number of more efficient sparse-dense inner products. Our mechanism for doing this is described in Section 1.2.

1.2. Matrix coloring. Matrix coloring is related to graph coloring, an important topic in graph theory [7]. We are not interested in a graph-theoretic understanding, only in the specific use of graph theory to color matrices, so we introduce only terms specifically relevant to our usage.

DEFINITION 1.1. *Let $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$. The vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ are structurally orthogonal if $|\boldsymbol{u}|^T |\boldsymbol{v}| = 0$, where $|\boldsymbol{u}|$ is the vector of the absolute value of all the elements of $\boldsymbol{u}$. A set of vectors $\{\boldsymbol{u}_1, ..., \boldsymbol{u}_n\}$ is called structurally orthogonal if $\boldsymbol{u}_i$ and $\boldsymbol{u}_j$ are structurally orthogonal for $1 \leq i, j \leq n$.*

Under this definition, not all orthogonal vectors are structurally orthogonal; for example,

$$\boldsymbol{u} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \qquad \boldsymbol{v} = \begin{pmatrix} 1 \\ -1 \end{pmatrix},$$

so $\boldsymbol{u}^T \boldsymbol{v} = 0$ but $|\boldsymbol{u}|^T |\boldsymbol{v}| = 2$. Also, all vectors are structurally orthogonal to a vector of all zeros.

LEMMA 1.2. *Let $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$ be structurally orthogonal, and denote $\boldsymbol{u}(k)$ as the kth element in $\boldsymbol{u}$. Then for $1 \leq k \leq n$, either $\boldsymbol{u}(k) = 0$ or $\boldsymbol{v}(k) = 0$, or both.*

Proof. We know that $|\boldsymbol{u}|^T |\boldsymbol{v}| = 0$, which means

$$\sum_{k=1}^{n} |\boldsymbol{u}(k)||\boldsymbol{v}(k)| = 0.$$

Obviously, $|\boldsymbol{u}(k)||\boldsymbol{v}(k)| \geq 0$ for $1 \leq k \leq n$; for the sum of these nonnegative terms to equal zero, all the terms must themselves be zero:

$$|\boldsymbol{u}(k)||\boldsymbol{v}(k)| = 0, \qquad 1 \leq k \leq n.$$

Therefore, either $\boldsymbol{u}(k) = 0$ or $\boldsymbol{v}(k) = 0$, or potentially both. $\square$

DEFINITION 1.3. *Let $\mathsf{C} \in \mathbb{R}^{m \times n}$. An* orthogonal set *is a set of q indices $\boldsymbol{\ell} = \{\ell^1, \ell^2, \dots, \ell^q\}$ for which $\mathsf{C}(:, \ell^i)$ and $\mathsf{C}(:, \ell^j)$ are structurally orthogonal when $1 \leq i, j \leq n$ and $i \neq j$. We will also define a set containing only one index $\boldsymbol{\ell} = \{\ell^1\}$ to be an* orthogonal set.

DEFINITION 1.4. *Let $\mathsf{C} \in \mathbb{R}^{m \times n}$. A* matrix coloring *$c = \{\boldsymbol{\ell}_1, \boldsymbol{\ell}_2, \dots, \boldsymbol{\ell}_p\}$ is a collection of index sets such that, for $1 \leq k \leq n$, the index k appears in exactly one index set. We say that c is a* valid matrix coloring *of C if each index set in c is an orthogonal set of C. We refer to an orthogonal set that is part of a coloring as a* color.

Because we require $1 \leq k \leq n$ to appear in exactly one orthogonal set, every column of C appears in exactly one color. Each orthogonal set in the coloring contains a set of indices corresponding to columns of C which are structurally orthogonal. The term coloring mimics the graph coloring concept of grouping nonadjacent vertices using the same color; here we are grouping structurally orthogonal columns of a matrix using the same orthogonal set.

Recall that we allow for the possibility of there being only one column in a color; for instance, a column with no zero values in it must exist in its own color. This in turn guarantees that every matrix has a coloring, since every matrix $\mathsf{C} \in \mathbb{R}^{m \times n}$ must have at least the trivial coloring $\{\{1\}, \{2\}, \dots, \{n\}\}$. If the matrix C were totally dense, with no structural zeros at all, this would be the only coloring. Our focus, however, is on the coloring of very sparse matrices.

A matrix may have more than one valid coloring; the 2×2 identity

$$\mathsf{I}_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

has two colorings: $c_1 = \{\{1, 2\}\}$ and $c_2 = \{\{1\}, \{2\}\}$. We are not concerned with how the coloring of a matrix is determined but, rather, how we can use it to efficiently compute matrix-matrix products. Therefore, we refer readers to [3] for an introduction to how colorings are found.

DEFINITION 1.5. *Let $\mathsf{C} \in \mathbb{R}^{m \times n}$ and $c = \{\boldsymbol{\ell}_1, \dots, \boldsymbol{\ell}_p\}$ be a valid matrix coloring for C. Applying the coloring c to C produces a matrix C_{Dense} that is at least as dense as the matrix C. If c has p colors in it, the matrix $\mathsf{C}_{Dense} \in \mathbb{R}^{m \times p}$. The kth column of the matrix C_{Dense} is created by combining the columns of C from the orthogonal set $\boldsymbol{\ell}_k = \{\ell_k^1, \dots, \ell_k^{q_k}\}$. Specifically,*

$$\mathsf{C}_{Dense}(j, k) = \begin{cases} \mathsf{C}(j, \ell_k^r), & \text{if} & \mathsf{C}(j, \ell_k^r) \neq 0, \\ 0, & \text{if} & \sum_{i=1}^{q_k} |\mathsf{C}(j, \ell_k^i)| = 0, \end{cases}$$

for $1 \leq j \leq m$. The matrix resulting from applying the coloring will be referred to as a compressed matrix.

THEOREM 1.6. *Let $\mathsf{C} \in \mathbb{R}^{m \times n}$ and $c = \{\boldsymbol{\ell}_1, \dots, \boldsymbol{\ell}_p\}$ be a valid matrix coloring for C. The matrix C_{Dense} created by applying c to C is unique.*

Proof. Each column of C_{Dense} is created from a single color, and no column of C appears in more than one color. Thus, proving that any column of C_{Dense} is unique proves that C is unique. We will prove that each value of the kth column of C_{Dense} is unique.

If the value $\mathsf{C}_{Dense}(j,k) = 0$, then $\sum_{i=1}^{q_k} |\mathsf{C}(j,\ell_k^i)| = 0$, which could occur only if $\mathsf{C}(j,\ell_k^1) = \ldots = \mathsf{C}(j,\ell_k^{q_k}) = 0$. This would prevent having a conflicting nonzero value for that location, so all zero values in the kth column are unique.

If the value $\mathsf{C}_{Dense}(j,k) \neq 0$, then at least one of the values in $\{\mathsf{C}(j,\ell_k^1),\ldots,\mathsf{C}(j,\ell_k^{q_k})\}$ is nonzero. If exactly one value is nonzero, then the value $\mathsf{C}_{Dense}(j,k)$ is unique. We must prove that only one of these q_k values is nonzero; we begin by assigning the nonzero value to the ν_k index, namely, $\mathsf{C}_{Dense}(j,\ell_k^{\nu_k}) \neq 0$.

Because c is a valid matrix coloring of C, the columns $\{\mathsf{C}(:,\ell_k^1),\ldots,\mathsf{C}(:,\ell_k^{q_k})\}$ must be structurally orthogonal. Lemma 1.2 tells us that for the jth row,

$$\mathsf{C}(j,\ell_k^s)\mathsf{C}(j,\ell_k^{\nu_k}) = 0, \qquad 1 \leq s \leq q_k, \quad s \neq \nu_k.$$

Since we have assumed that $\mathsf{C}_{Dense}(j,\ell_k^{\nu_k}) \neq 0$, this lemma demands that $\mathsf{C}(j,\ell_k^s) = 0$ for $1 \leq s \leq q_k$, $s \neq \nu_k$. Therefore, every value in the kth column of C_{Dense} is uniquely determined. $\square$

Traditionally, matrix colorings have been used for the accelerated computation of finite-difference Jacobians for the purpose of preconditioning nonlinear solvers. In that setting, structurally orthogonal columns were grouped together to allow for simultaneous function evaluation. This minimizes the number of function calls required to approximate the Jacobian without ignoring any nonzero values. We use this same concept in Section 2 to accelerate the operation $\mathsf{C} = \mathsf{A}\mathsf{B}^T$.

2. Analysis of Coloring for Sparse Matrix-Matrix Products.

Our goal in this section is to exploit the graph coloring concept, described in Section 1.2, to more efficiently compute inner products in a sparse-sparse matrix product. We are interested in the two expressions

$$\mathsf{C} = \mathsf{A}\mathsf{B}^T \text{ or} \tag{2.1a}$$

$$\mathsf{C} = \mathsf{R}\mathsf{A}\mathsf{R}^T, \tag{2.1b}$$

where matrices $\mathsf{A}, \mathsf{B}, \mathsf{R}$ are stored in CSR format and are not necessarily square. We analyze only (2.1a) in this section because (2.1b) can be described by using this product.

The sparse-sparse matrix product (2.1a) can be computed with an analagous sparse-dense matrix product; doing so will reduce memory access overhead and improve computational efficiency. To achieve this improved performance, we restructure the sparse matrix B^T into a dense matrix B_{Dense}^T using the matrix coloring ideas introduced in Section 1.2. This allows us to compute

$$\mathsf{C}_{Dense} = \mathsf{A}\mathsf{B}_{Dense}^T, \tag{2.2}$$

such that the nonzero entries in C_{Dense} are also the nonzero entries needed to form C. After computing (2.2), C_{Dense} must be reorganized into C, which is done by using a process similar to the compression of B^T into B_{Dense}^T. Since this is not fundamental to the analysis, this process will be discussed in Section 3.1.

We must ask the question: How can we form a dense B_{Dense}^T such that C_{Dense} and C have the same nonzero values? Our approach uses a coloring of the matrix C to determine which columns of C are structurally orthogonal and can be computed simultaneously. This allows for the compression of those multiple columns associated with each color of B^T into a single column of B_{Dense}^T. After C_{Dense} has been computed, this single dense column is decompressed to fill all the columns of C corresponding to the columns of B^T, which were earlier compressed to created B_{Dense}^T.

Although the dense matrix C_{Dense} will contain the same nonzero values as C, the dense matrix, B_{Dense}^T will not necessarily contain all the nonzero values of B^T. This special case may occur when A has columns with only zero values, but we prove in Corollary 2.7 that whenever A has no zero columns, any coloring of C *will be a valid coloring* for B^T.

2.1. Motivating Example.
We consider a small example to demonstrate the coloring concept, before we study its validity in general sparse-sparse matrix products. Consider the product of two "sparse" matrices,

$$
\underbrace{\begin{pmatrix}
1 & 6 & 3 & & & 12 \\
 & 4 & & & & 12 \\
1 & 2 & 3 & & & 36 \\
 & 12 & 28 & 15 & & 24 \\
 & & 20 & & & 66 \\
 & & & & & 36
\end{pmatrix}}_{\mathsf{C}}
=
\underbrace{\begin{pmatrix}
1 & 2 & & & & \\
 & 2 & & & & \\
1 & & & & & 6 \\
 & & 3 & 4 & & \\
 & & & & 5 & 6 \\
 & & & & & 6
\end{pmatrix}}_{\mathsf{A}}
\underbrace{\begin{pmatrix}
1 & 2 & 3 & & & \\
 & 2 & & & & 6 \\
 & & & 4 & 2 & \\
 & & 3 & 4 & & 6 \\
 & & & 4 & & 6 \\
 & & & & & 6
\end{pmatrix}}_{\mathsf{B}^T}.
$$

The structure of C admits the coloring

$$c = \{\boldsymbol{\ell}_1, \boldsymbol{\ell}_2, \boldsymbol{\ell}_3, \boldsymbol{\ell}_4\} = \{\{1,4\}, \{2,5\}, \{3\}, \{6\}\}$$

because columns 1 and 4 are structurally orthogonal, as are columns 2 and 5. Notice that despite the orthogonality of the third and fifth columns of B^T, those columns cannot be combined in the same color in C. Using the coloring c, we can compress B^T into B^T_{Dense} and compute C_{Dense},

$$
\underbrace{\begin{pmatrix}
1 & 6 & 3 & 12 \\
 & 4 & & 12 \\
1 & 2 & 3 & 36 \\
28 & 15 & 12 & 24 \\
20 & & & 66 \\
 & & & 36
\end{pmatrix}}_{\mathsf{C}_{Dense}}
=
\underbrace{\begin{pmatrix}
1 & 2 & & & & \\
 & 2 & & & & \\
1 & & & & & 6 \\
 & & 3 & 4 & & \\
 & & & & 5 & 6 \\
 & & & & & 6
\end{pmatrix}}_{\mathsf{A}}
\underbrace{\begin{pmatrix}
1 & 2 & 3 & \\
 & 2 & & 6 \\
4 & 5 & & \\
4 & & 3 & 6 \\
4 & & & 6 \\
 & & & 6
\end{pmatrix}}_{\mathsf{B}^T_{Dense}}.
$$

To decompress C_{Dense} into C, we must store both the columns that formed the coloring and the rows of C associated with each of those columns. In the example above, the first column of C_{Dense} is composed of columns 1 and 4 from C, and the fourth and fifth rows belong to column 4 of C. This process is discussed in Section 3.1.

2.2. Proof of the Validity of Matrix Coloring. For a coloring of C to be appropriate for the product AB^T, one of two situations must apply:

1. The coloring c of C is also a valid coloring of B^T; invoking Theorem 1.6 means that a unique B^T_{Dense} exists when applying c, or
2. The conflicts that arise when applying the coloring do not contribute to the nonzeros of C.

We must prove this second condition; before we can complete such a proof, we need to create a new device: an auxiliary matrix $\hat{\mathsf{B}}^T$ for which c is a valid coloring.

DEFINITION 2.1. *Let $\mathsf{B}^T \in \mathbb{R}^{r \times n}$ and c be a coloring for a matrix with n columns. A* vacated matrix $\hat{\mathsf{B}}^T \in \mathbb{R}^{r \times n}$ *is a matrix whose nonzero values all coincide with nonzeros from B^T but for which c is a valid coloring.*

Under this definition, no nonzeros can be introduced in vacating B^T to $\hat{\mathsf{B}}^T$; values can only be zeroed out to produce a matrix with the appropriate structure to apply the coloring. If c is a valid coloring of B^T, then one trivial vacation of B^T would be to remove no nonzeros. Another trivial vacation of B^T that would validate any coloring would be to zero out every value in the matrix.

DEFINITION 2.2. *Let $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_q \in \mathbb{R}^r$. The* conflicted index set *of $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_q\}$ is the set of indices*

$$\Gamma(\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_q\}) = \{\, \gamma \in \{1, 2, \ldots, m\} \;\mid\; \boldsymbol{u}_i(\gamma)\boldsymbol{u}_j(\gamma) \neq 0, \quad \text{for some} \quad 1 \leq i, j \leq q, \quad i \neq j\}.$$

We define the conflicted index set of a set of one vector as empty: $\Gamma(\{\boldsymbol{u}\}) = \emptyset$.

The term *conflicted index set* refers to the fact that these rows are preventing the set of vectors $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_q\}$ from being structurally orthogonal. Were they structurally orthogonal, then the conflicted index set would be empty.

LEMMA 2.3. *Let $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_q \in \mathbb{R}^r$, and $\Gamma(\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_q\})$ be the associated conflicted index set. The set of vectors $\{\hat{\boldsymbol{u}}_1, \hat{\boldsymbol{u}}_2, \ldots, \hat{\boldsymbol{u}}_q\}$ defined as*

$$
\hat{\boldsymbol{u}}_i(\gamma) = \begin{cases} 0 & \gamma \in \Gamma(\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_q\}) \\ \boldsymbol{u}_i(\gamma) & else \end{cases}
$$

is structurally orthogonal.

Proof. If $\Gamma = \emptyset$, then the vectors in $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_q\}$ are already structurally orthogonal. Otherwise, we must prove that

$$|\hat{\boldsymbol{u}}_i|^T |\hat{\boldsymbol{u}}_j| = 0, \qquad 1 \le i, j \le q, \qquad i \ne j.$$

Let us simplify the notation for this proof by defining $\Gamma \equiv \Gamma(\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_q\})$. For any $i \ne j$, the inner product can be separated into two components,

$$|\hat{\boldsymbol{u}}_i|^T |\hat{\boldsymbol{u}}_j| = \sum_{\gamma \in \Gamma} |\hat{\boldsymbol{u}}_i(\gamma)||\hat{\boldsymbol{u}}_j(\gamma)| + \sum_{\gamma \notin \Gamma} |\hat{\boldsymbol{u}}_i(\gamma)||\hat{\boldsymbol{u}}_j(\gamma)|$$

$$= \sum_{\gamma \in \Gamma} |0||0| + \sum_{\gamma \notin \Gamma} |\boldsymbol{u}_i(\gamma)||\boldsymbol{u}_j(\gamma)| = \sum_{\gamma \notin \Gamma} |\boldsymbol{u}_i(\gamma)||\boldsymbol{u}_j(\gamma)|.$$

Because the conflicted index set includes indices such that $\boldsymbol{u}_i(\gamma)\boldsymbol{u}_j(\gamma) \ne 0$, any $\gamma \notin \Gamma$ must have $\boldsymbol{u}_i(\gamma)\boldsymbol{u}_j(\gamma) = 0$, leaving the summation above equal to zero. $\square$

This lemma allows us to take any set of vectors and replace certain nonzero values with zeros to make them structurally orthogonal. We will apply this concept to the columns of B^T involved in each color of c in order to create a matrix $\hat{\mathsf{B}}^T$ for which c is a valid coloring.

DEFINITION 2.4. *Let* $\mathsf{B}^T \in \mathbb{R}^{r \times n}$, $c = \{\boldsymbol{\ell}_1, \ldots, \boldsymbol{\ell}_p\}$ *be a coloring for some matrix with n columns, and let* $\mathsf{B}^T(:, \boldsymbol{\ell}_k)$ *denote the set of columns associated with the color* $\boldsymbol{\ell}_k = \{\ell_k^1, \ldots, \ell_k^{q_k}\}$. *The* minimally vacated *matrix* $\hat{\mathsf{B}}_c^T \in \mathbb{R}^{r \times n}$ *is the vacated matrix generated from* B^T *in the following way:*

$$\hat{\mathsf{B}}_c^T(\gamma, \ell_k^j) = \begin{cases} 0, & \gamma \in \Gamma(\mathsf{B}^T(:, \boldsymbol{\ell}_k)) \\ \mathsf{B}^T(\gamma, \ell_k^j), & else \end{cases}, \qquad 1 \le j \le q_k, \quad 1 \le k \le p.$$

LEMMA 2.5. *Let* $\mathsf{B}^T \in \mathbb{R}^{r \times n}$, $c = \{\boldsymbol{\ell}_1, \ldots, \boldsymbol{\ell}_p\}$ *be a valid coloring for some matrix with n columns. The coloring c is valid for the minimally vacated matrix* $\hat{\mathsf{B}}_c^T$.

Proof. Using Lemma 2.3, we know that the columns in $\hat{\mathsf{B}}_c^T(:, \boldsymbol{\ell}_k)$ are structurally orthogonal for $1 \le k \le p$, and therefore c is a valid coloring of $\hat{\mathsf{B}}_c^T$. $\square$

The minimal vacation of a matrix allows for the preservation of nonzeros with which there is no coloring conflict. Recall that for the product $\mathsf{C} = \mathsf{A}\mathsf{B}^T$, our goal is to produce a dense matrix B_{Dense}^T from the matrix B^T by applying the coloring c to B^T. As stated earlier, c may not be a valid coloring for B^T, in which case we substitute the minimally vacated matrix $\hat{\mathsf{B}}_c^T$ in place of B^T. Lemma 2.5 proves that this c is an acceptable coloring for this vacated matrix.

More important than the validity of the coloring is the validity of the equation $\mathsf{C} = \mathsf{A}\hat{\mathsf{B}}_c^T$. If this equality does not hold, then some values in C would be altered by substituting the minimally vacated $\hat{\mathsf{B}}_c^T$ for B^T, which is unacceptable. Theorem 2.6 addresses this concern.

THEOREM 2.6. *Let* $\mathsf{A} \in \mathbb{R}^{m \times r}$, $\mathsf{B} \in \mathbb{R}^{n \times r}$ *and* $c = \{\boldsymbol{\ell}_1, \ldots, \boldsymbol{\ell}_p\}$ *be a coloring of* $\mathsf{C} = \mathsf{A}\mathsf{B}^T$. *If* $\hat{\mathsf{B}}_c^T$ *is the minimally vacated matrix generated by applying c to the matrix* B^T, *then the equality*

$$\mathsf{A}\mathsf{B}^T = \mathsf{A}\hat{\mathsf{B}}_c^T \tag{2.3}$$

holds.

Proof. We denote $\hat{\mathsf{C}} = \mathsf{A}\hat{\mathsf{B}}_c^T$ and prove that $\mathsf{C} = \hat{\mathsf{C}}$ by proving that all the nonzeros of $\hat{\mathsf{C}}$ match the nonzeros of C. By definition, each column of C belongs to exactly one color, so let us denote as $\boldsymbol{\ell}_{p_j} = \{\ell_{p_j}^1, \ldots, j, \ldots, \ell_{p_j}^{q_j}\}$ the color containing column $\mathsf{C}(:, j)$. As was done earlier, let $\mathsf{C}(:, \boldsymbol{\ell}_{p_j})$ denote the set of columns of C associated with the color $\boldsymbol{\ell}_{p_j}$.

Call $\Gamma_j \equiv \Gamma(\mathsf{B}^T(:, \boldsymbol{\ell}_{p_j}))$ the conflicted index arising from applying the color $\boldsymbol{\ell}_{p_j}$ to the matrix B^T. The values $\mathsf{C}(i, j)$ and $\hat{\mathsf{C}}(i, j)$ can be computed in pieces related to this Γ_j:

$$\mathsf{C}(i, j) = \sum_{\gamma \in \Gamma_j} \mathsf{A}(i, \gamma)\mathsf{B}^T(\gamma, j) + \sum_{\gamma \notin \Gamma_j} \mathsf{A}(i, \gamma)\mathsf{B}^T(\gamma, j),$$

$$\hat{\mathsf{C}}(i, j) = \sum_{\gamma \in \Gamma_j} \mathsf{A}(i, \gamma)\hat{\mathsf{B}}_c^T(\gamma, j) + \sum_{\gamma \notin \Gamma_j} \mathsf{A}(i, \gamma)\hat{\mathsf{B}}_c^T(\gamma, j).$$

Using the definition of a minimally vacated matrix, we can simplify this second line to

$$\hat{\mathsf{C}}(i,j) = \sum_{\gamma \notin \Gamma_j} \mathsf{A}(i,\gamma) \mathsf{B}^T(\gamma, j),$$

which, combined with the first line, gives

$$\mathsf{C}(i,j) = \sum_{\gamma \in \Gamma_j} \mathsf{A}(i,\gamma) \mathsf{B}^T(\gamma, j) + \hat{\mathsf{C}}(i,j).$$

Since we want $\mathsf{C}(i,j) = \hat{\mathsf{C}}(i,j)$, we must prove that

$$\sum_{\gamma \in \Gamma_j} \mathsf{A}(i,\gamma) \mathsf{B}^T(\gamma, j) = 0, \qquad 1 \le i \le m, \qquad 1 \le j \le n,$$

which we do by proving that

$$\text{for all } 1 \le i \le m, \ 1 \le j \le n : \quad \mathsf{A}(i,\gamma) = 0 \ \text{ for each } \ \gamma \in \Gamma_j. \tag{2.4}$$

By the definition of the conflicted index set Γ_j, for every $\gamma \in \Gamma_j$ at least two columns of B^T must have nonzero rows γ. If we choose any two of those and call their indexes s and t, we can write

$$\gamma \in \Gamma_j \quad \Rightarrow \quad \text{there exists} \quad s,t \in \boldsymbol{\ell}_{p_j}, \quad s \ne t \quad \text{such that} \quad \mathsf{B}^T(\gamma, s) \ne 0, \text{ and } \mathsf{B}^T(\gamma, t) \ne 0. \tag{2.5}$$

Because $\boldsymbol{\ell}_{p_j}$ is a valid color for C, the set $\mathsf{C}(:, \boldsymbol{\ell}_{p_j})$ is structurally orthogonal:

$$\mathsf{C}(i,s) = 0, \text{ or } \mathsf{C}(i,t) = 0, \qquad 1 \le i \le m.$$

The structural zero $\mathsf{C}(i,s)$ occurs because the vectors $\mathsf{A}(i,:)^T$ and $\mathsf{B}^T(:,s)$ are structurally orthogonal. Applying Lemma 1.2 gives

$$\mathsf{A}(i,\gamma) = 0, \text{ or } \mathsf{B}^T(\gamma, s) = 0, \qquad 1 \le \gamma \le r, \qquad 1 \le i \le m.$$

A similar statement is also true for $\mathsf{C}(i,t)$, so we join these results to state that for every $1 \le \gamma \le r$,

$$\text{either } \mathsf{B}^T(\gamma, s) = 0 \text{ or } \mathsf{B}^T(\gamma, t) = 0, \text{ or } \mathsf{A}(i,\gamma) = 0 \text{ for all } 1 \le i \le m. \tag{2.6}$$

The combination of (2.5) and (2.6), along with the knowledge that each column of C appears in only one color, is sufficient to prove (2.4). □

Theorem 2.6 guarantees that the coloring of C is a sufficient tool to perform the sparse-dense matrix product AB^T_{Dense}. While creating the minimally vacated matrix $\hat{\mathsf{B}}^T_c$ is not difficult in practice, most applications do not require it, as shown in Corollary 2.7.

COROLLARY 2.7. *For the product $\mathsf{C} = \mathsf{AB}^T$, any coloring c of C is a valid coloring of B^T if A has no zero columns.*

Proof. Start with (2.6), which is valid for any two distinct columns $s,t \in \boldsymbol{\ell}_{p_j}$. If A has no zero columns, then $\mathsf{A}(i,\gamma) \ne 0$ for some $1 \le i \le m$, which requires that either $\mathsf{B}^T(\gamma, s) = 0$ or $\mathsf{B}^T(\gamma, t) = 0$ for $1 \le \gamma \le r$ and makes c a valid coloring for B^T. □

3. Algorithms for Sparse-Sparse Matrix Product Using Coloring. Now that we have laid the foundation, we present Algorithm 3 for computing the product of sparse matrices by an associated sparse-dense product generated through matrix coloring. The remainder of this section will analyze this algorithm and adapt it to address implementation concerns.

One way to compare Algorithm 3 to a sparse inner product-based algorithm would be to study the memory traffic. Let n_A and n_B denote the average number of nonzeros per row in A and B, respectively, and let n_{color} be the number of colors in c. From our discussion in Section 1.1, we can surmise that computing AB^T using sparse inner products requires roughly $O((n_\mathsf{A} + n_\mathsf{B})mn)$ memory accesses. On the other hand, AB^T_{Dense} should incur only $O(2n_\mathsf{A}mn_{color})$ memory accesses, which could yield substantial savings if $n_{color} \ll n$.

Algorithm 3 Computing $\mathsf{C} = \mathsf{AB}^T$ Using Coloring, Basic Version

---

1: Compute symbolic $\mathsf{C} = \mathsf{AB}^T$; Compute a matrix coloring of C, c
2: Assemble B_{Dense}^T by applying c to B^T
3: Perform the sparse-dense matrix product $\mathsf{C}_{Dense} = \mathsf{AB}_{Dense}^T$
4: Recover C from C_{Dense}

Despite the gains from performing a sparse-dense matrix product, costs are incurred by computing the coloring of C, compressing B^T to B_{Dense}^T and then recovering C from C_{Dense}. These eat into the competitive advantage described earlier for the sparse-dense matrix product and must be taken into account when comparing the two algorithms. Rather than try to conduct a "pencil and paper" analysis of this additional complexity, we have performed numerical tests to demonstrate that their cost is not overwhelming. Those results appear in Section 4.

In line 1 of Algorithm 3, we compute the symbolic product of AB^T, which determines the location of the nonzeros in C before their value is computed [10]. Generally, this is used to preallocate space for C, but here we also use this structural description of C to determine a matrix coloring c. We will not discuss the different algorithms for computing matrix and graph colorings here; the effect of two popular choices, which are available in the PETSc library, are compared in Section 4.2.

Line 2 was discussed in Section 2, although here we omit the possible need for a vacated matrix $\hat{\mathsf{B}}^T$ to form B_{Dense}^T. Recall that Corollary 2.7 shows that this vacation is not necessary so long as A has at least one nonzero in each column. The sparse-dense matrix product in line 3 is computed as a sequence of sparse-dense inner products, each of which implements Algorithm 2. The efficient recovery of the sparse matrix C from the computed matrix C_{Dense} is not trivial; it is discussed in Section 3.1.

Lines 2 through 4 contain the so-called numeric portion of the matrix product. For many applications, including the PFLOTRAN example in Section 4.1, the nonzero patterns of matrices remain fixed despite varying numeric values; this allows the symbolic component (line 1) to be performed only once while numeric products are computed whenever matrix values are updated. Our efficiency discussions focus on the numeric component.

Although not the focus of this paper, performing the sparse-dense product AB_{Dense}^T in lieu of the sparse-sparse product AB^T allows for new optimizations leveraging the standardized structure of a dense matrix. One optimization that we exploit is the computation of multiple columns of C_{Dense} simultaneously. Each time A is brought into memory, four columns of C_{Dense} are computed, allowing for more flops per cache miss caused by A. The number of columns that can be efficiently computed simultaneously is determined by the available memory. Improving the ratio of flops per cache miss was a key factor in motivating this work, as discussed in Section 1.1, and the topic will appear again in Section 4.

3.1. Recovering the Sparse Matrix from the Dense Product. The fourth line of Algorithm 3 decompresses the dense matrix C_{Dense} to the sparse matrix C. Although we have not explicitly stated it previously, more information is needed to perform this decompression than just C_{Dense} and c. In the process of compressing B^T to B_{Dense}^T, multiple columns (previously denoted $\mathsf{B}^T(:, \boldsymbol{\ell}_j)$) are joined to form a single column $\mathsf{B}_{Dense}^T(:, j)$; to undo this process, we must know how to partition $\mathsf{C}_{Dense}(:, j)$ among the columns in the set $\mathsf{C}(:, \boldsymbol{\ell}_j)$.

In practice, during the compression, the coloring is augmented to also store this row data. Referring to the example in Section 2.1, we would augment the coloring

$$c = \{\{1, 4\}, \{2, 5\}, \{3\}, \{6\}\}$$

to include the following row data:

$$c^+ = \{\{1 : \{1, 3\}, 4 : \{4, 5\}\}, \{2 : \{1, 2, 3\}, 5 : \{4\}\}, \{3 : \{1, 3, 4\}\}, \{6 : \{1, 2, 3, 4, 5, 6\}\}\}.$$

Using this augmented coloring c^+, we can decompress a compressed matrix to its sparse form.

The strategy used to populated C from C_{Dense} can contribute significantly to the computational cost of Algorithm 3. A simple approach would be to simply traverse C_{Dense} contiguously and populate C with each nonzero according to c^+. Implementation of this revealed that as the matrix sizes increase, the decompression

can consume up to $1/3$ of the total execution time; in contrast, the compression of B^T to B^T_{Dense} generally requires a much smaller portion of the total time.

The most direct implementation of the C_{Dense} decompression ignores the fact that values that are very close in C_{Dense} may be in very distant columns of C. This occurs because C_{Dense} is stored in dense *columns* (as is the dense matrix storage standard) but C is a compressed sparse *row* matrix. Unpacking any single column of C_{Dense} may insert nonzero entries throughout C; therefore, decompressing each column of C_{Dense} may cause a traversal through the entire C matrix, which will likely incur excessive cache misses.

To mitigate this expense, we could fill some block of rows of C all at once, thereby preventing the need for n_{color} passes through C. Results for both decompression techniques are presented in Section 4.1. Algorithm 4 incorporates the changes discussed in this subsection into the coloring-based sparse product algorithm and also notes the potential need for a minimally vacated B^T as described in Theorem 2.6. This algorithm is listed for completeness, to indicate the practical steps that must be taken for implementation; we will in general refer instead to Algorithm 3 for simplicity.

Algorithm 4 Computing $\mathsf{C} = \mathsf{AB}^T$ Using Coloring, Practical Version

---

1: Compute symbolic $\mathsf{C} = \mathsf{AB}^T$
2:      Compute c, a matrix coloring of C
3:      Vacate B^T using c if needed for compression
4: Assemble B^T_{Dense} by applying c to B^T
5:      Augment matrix coloring $c \rightarrow c^+$
6: Perform the sparse-dense matrix product $\mathsf{C}_{Dense} = \mathsf{AB}^T_{Dense}$
7: Recover C from C_{Dense} using c^+
8:      Populate $m_{blocksize}$ rows of C at once

3.2. Algorithms for the RAR^T Product. Thus far we have discussed the product (2.1a), but our motivating application is multigrid and it involves the product (2.1b). When Algorithm 3 is adapted for the product $\mathsf{C} = \mathsf{RAR}^T$, two options arise: C can be computed by using two sparse-dense products or one sparse-dense product and one sparse-sparse product. The relative efficiency of these options is affected by the number of colors present in the compression.

The auxiliary matrix W allows us to compute (2.1b) in two steps:

$$W = AR^T, \tag{3.1a}$$
$$C = RW. \tag{3.1b}$$

We will always compute (3.1a) using coloring, but the choice of coloring will determine the efficiency of that and subsequent computations. The coloring of W, c_W, can be used to implement Algorithm 3, at the end of which, a sparse W is returned. Then a sparse-sparse matrix product between R and W is used to compute C. This algorithm is described in Algorithm 5.

Algorithm 5 Computing $\mathsf{C} = \mathsf{RAR}^T$ Using the Coloring of AR^T

1: Use Algorithm 3 to compute $\mathsf{W} = \mathsf{AR}^T$
2: Compute the sparse-sparse CSR matrix product $\mathsf{C} = \mathsf{RW}$

In line 2 of this algorithm, a sparse-sparse matrix product is used instead of the sparse-dense matrix product that we have been developing in this paper. This is counterintuitive because we have been discussing the excessive cost of performing sparse-sparse products, but it may be preferable depending on the number of colors present in C. If C has too many colors, then it will be faster to use a sparse product to compute it, as discussed in Section 3. Should C have few colors, it may be faster to use Algorithm 6 to compute C.

Algorithm 6 involves two sparse-dense matrix products, each using the coloring c_C instead of the coloring c_W as was used in Algorithm 5. The cost of this algorithm is tied to the number of colors in C, which can be greater than the number of colors in W; for many of the multigrid problems we study, C is much more dense than W, and it will be less effective to use c_C for these products. Computational results presented in

---

Algorithm 6 Computing $\mathsf{C} = \mathsf{RAR}^T$ Using the Coloring of C

1: Compute symbolic $\mathsf{C} = \mathsf{RAR}^T$; Compute matrix coloring c_{C}
2:     Compress (and vacate, if needed) R^T with c_{C} to form R^T_{Dense}
3:     Augment c_{C} to c_{C}^+ with the necessary row information
4: Use a sparse-dense matrix product to compute $\mathsf{W}_{Dense} = \mathsf{AR}^T_{Dense}$
5: Use a sparse-dense matrix product to compute $\mathsf{C}_{Dense} = \mathsf{RW}_{Dense}$
6: Recover C from C_{Dense} using c_{C}^+

Section 4.1 compare these two algorithms and show that the number of colors present in the coloring is a major factor in the efficiency of the computation.

4. Numerical Experiments. We present two sets of test cases:
- The regional doublet test case from PFLOTRAN [8] and
- A three-dimensional linear elasticity PDE test provided in the PETSc library distribution [2].

These tests were chosen because algebraic multigrid is an efficient solver for the linear systems arising in both applications.

Multigrid is a mathematically optimal method for solving the system of algebraic equations that arise from discretized elliptic boundary value PDEs [12, 13]. We use a geometric-algebraic multigrid method (GAMG) in PETSc, which integrates geometric information into robust algebraic multigrid formulations to yield superior convergence rates of the multigrid solver [1, 2].

GAMG requires the matrix triple products $\mathsf{C} = \mathsf{RAR}^T$ to be computed on all grid levels in the solver setup phase and the matrix product $\mathsf{C} = \mathsf{GG}^T$ to be computed for creation of connection graphs. These matrix products, as common computational primitives, constitute a significant portion of the entire simulation cost.

The experiments were conducted by using the PETSc library on a Dell Poweredge 1950 server with dual Intel Xeon E5440 quadcore CPUs at 2.83 GHz and 16 GB DDR2-667 memory in 4 channels providing 21 GB/s total memory bandwidth. The machine runs Ubuntu Linux 12.04 64 bit OS. The execution time and floating-point rates were obtained by using one core with the GNU compiler version 4.7.3 and -O optimization. Our performance results were obtained by running the entire test cases and profiling the relevant matrix products. We have done so because standalone benchmarking often produces unreasonably optimistic reports since much of the data is already in cache, which is not the case during an actual simulation.

4.1. Regional Doublet Test Case from PFLOTRAN. We demonstrate the use of matrix coloring on the regional doublet test case [5] from PFLOTRAN, a state-of-the-art code for simulating multiscale, multiphase, multicomponent flow and reactive transport in geologic media. PFLOTRAN solves a coupled system of mass and energy conservation equations for a number of phases, including air, water, supercritical CO_2, and a number of chemical components. PFLOTRAN is built on the PETSc library and makes extensive use of PETSc iterative nonlinear and linear solvers.

The regional doublet test case models variable saturated goundwater flow and solute transport within a hypothetical aquifer measuring 5000 m × 100 m. We consider only flow problems here because flow solves dominate computation. The governing equation is a system of time-dependant PDEs. PFLOTRAN utilizes finite-volume or mimetic finite-difference spatial discretizations and backward-Euler (fully implicit) timestepping. At each time step, Newton-Krylov methods are used to solve the resulting nonlinear algebraic equations. In all the experiments reported below we have run 35 time steps, which is the minimum needed to resolve the basic physics.

Tables 4.1 and 4.2 show the benefit of using matrix coloring to speed execution time for computing matrix triple products over small to large three-dimensional meshes. Three approaches are compared for computing $\mathsf{C} = \mathsf{RAR}^T$:
- Using no coloring
    - $\mathsf{P}^T\mathsf{AP}$ – stores $\mathsf{P} = \mathsf{R}^T$, and then computes C with sparse *outer* products,
    - RAP – stores $\mathsf{P} = \mathsf{R}^T$, and performs products between these three CSR matrices,
    - RAR^T – Algorithm 1 is used to compute each value in C,
- Using the coloring of RAR^T - Algorithm 6, and
- Using the coloring of AR^T - Algorithm 5.

10

Table 4.1: PFLOTRAN: $C = RAR^T$
Fine grid size: $50 \times 25 \times 10$; A: $12,500 \times 12,500$, R: $1,203 \times 12,500$
Times presented are in seconds. The use of coloring in the RAR^T computation is clearly beneficial. Also, the choice of coloring (studying the structure of RAR^T and applying Algorithm 5 or studying AR^T and applying Algorithm 6) plays major role in the algorithm efficiency.

	No Coloring			Coloring	
	$P^T AP$	RAP	RAR^T	RAR^T $(n_{color}=59)$	AR^T $(n_{color}=20)$
3 Symbolic	.0066	.016	.011	.019	.024
246 Numeric	1.30	1.46	2.39	1.51	.760
Total Time	1.31	1.48	2.40	1.53	.784

Table 4.2: PFLOTRAN: $C = RAR^T$
Total computation time (in seconds) presented for increasing fine grid density. The "time$_{100}$" column implements dense to sparse decompression with 100 row blocks. Algorithm 5 continues to outperform Algorithm 6. Unpacking multiple rows at once during the decompression (as discussed in Section 3.1) speeds the computations for the AR^T coloring but not for the RAR^T coloring.

	No Coloring	Coloring					
	RAR^T	Algorithm 6, using RAR^T			Algorithm 5, using AR^T		
Grid Size	time	n_{color}	time	time$_{100}$	n_{color}	time	time$_{100}$
$50 \times 25 \times 10$	2.4	59	1.5	1.6	20	.78	.84
$100 \times 50 \times 20$	28	70	26	27	24	14	12
$200 \times 100 \times 40$	246	84	374	376	25	162	132

The first column of Table 4.1 gives the total number of symbolic and numeric matrix triple products accumulated from all grid levels and all linear iterations during the entire simulation. The symbolic matrix products were computed only during the solver setup phase, while the numeric matrix products were executed for every nonlinear iteration of GAMG. Time spent creating matrix colorings is included in the symbolic row and contributed approximately half the symbolic execution time for the coloring columns. The matrix colorings for this example are created by using the PETSc default algorithm, largest-first ordering (discussed in Section 4.2). When compared with the repeated execution of numeric matrix products performed during the solve, the time spent on the symbolic products is minimal.

In Table 4.1, we see that the number of colors for matrix RAR^T is 59, whereas AR^T (Column 6) has only 20 colors. This disparity is the result of greater density in C. While both of these are far smaller than 1203, the column size of both R^T and the final sparse matrix product C, the greater sparsity in AR^T demands only a third as many colors and leads to significantly shorter execution time. The last row of Table 4.1 gives the total execution time, that is, the sum of the symbolic and numeric components. It shows that computing $C = RAR^T$ by using the matrix coloring of AR^T takes roughly one-third of the execution time of using no coloring.

Table 4.2 presents the same experiments over larger three dimensional meshes. The use of Algorithm 5 to compute the product continues to outperform standard sparse inner products and Algorithm 6. Here we also consider the dense to sparse decompression in block rows of 100 in addition to conducting the entire decompression in one sweep; this idea was introduced for $C = AB^T$ in Section 3.1, and the results are presented in the time$_{100}$ columns. There is a marked benefit when using the block decompression for the AR^T coloring option but no benefit for the RAR^T coloring. This can likely be attributed to the number of colors, since more colors indicates more zero-valued nonzeros during the dense compression and therefore extra work is performed.

Table 4.3 helps clear the computational picture by studying the flop rate for the three methods of computing RAR^T. The floating point computation required for the "No Coloring" option is significantly less than the coloring options because only nonzero matrix entries are involved in the computation; as discussed in Section 1.1, the efficiency (measured by flop rate) suffers as a result because many fewer flops take place

Table 4.3: PFLOTRAN: Flop Rate (megaflops/second)
Using coloring greatly increases the flop rate because of the sparse to dense compression. Even though the RAR^T coloring has a higher flop rate, its total computational time (from Table 4.2) is higher because many of the flops performed are unnecessary. The AR^T coloring has few enough colors in it to maintain a high flop rate without excessive unneeded computation involving zero-valued nonzeros. The coloring computation uses the 100 block row decompression.

Fine Meshes	No Coloring	Coloring RAR^T	Coloring AR^T
$50 \times 25 \times 10$	76	1232	724
$100 \times 50 \times 20$	64	936	606
$200 \times 100 \times 40$	59	663	496

per cache miss.

This issue is alleviated by the coloring because necessary values are stored contiguously in the dense matrix, thereby allowing for more efficient access. That improved efficiency is visible in the two columns of coloring results: both have significantly higher flop rates than the computation without coloring. Following this logic further, the higher flop rate would suggest that the RAR^T coloring is superior to the AR^T coloring. Although computations are happening more quickly there, we know that the total time required for the computation is also greater.

This seeming contradiction is caused by the number of zero-valued nonzeros introduced into the dense matrix. When the sparse matrices are used in the inner product computation, only nonzeros are involved in the computation, but accessing them is a slow proposition. On the other extreme, when the RAR^T coloring is used, too many zeros are involved in the inner products, allowing for more efficient memory access but involving so many zero-valued nonzeros that the total computational time is excessive. The coloring of AR^T has many fewer colors, which reduces the number of superfluous zeros in the dense matrix, and achieves the best overall performance. This is why the flop rate is slightly lower, but the total computational time (shown in Table 4.2) is better.

4.2. Three Dimensional Linear Elasticity PDEs. The set of test cases in this section can be found in the PETSc library (see petsc/src/ksp/ksp/examples/tutorials/ex56.c). They model a three-dimensional bilinear quadrilateral (Q1) displacement finite-element formulation of linear elasticity PDE defined over a unit cube domain with Dirichlet boundary condition on the side y=0, and load of 1.0 in x + 2y direction on all nodes. Three physical variables are defined at each grid point (i.e., degrees of freedom equals 3). We use the tests to further demonstrate the performance benefit of using matrix colorings and illustrate that the achieved acceleration depends on the matrix nonzero structures and selected matrix coloring.

We apply two matrix coloring algorithms: Largest-first ordering (LF) [6] and smallest-last ordering (SL) [9] to the grid operator matrices $\mathsf{C} = \mathsf{RAR}^T$ in Table 4.4 and to the matrix product $\mathsf{C} = \mathsf{GG}^T$ (used for connection graphs) in Table 4.5. These tables present the dimensions of the matrices, the execution time spent on computing numeric matrix products, and the number of colors obtained from the matrix colorings. We present only the results using the coloring of AR^T (Algorithm 5) in Table 4.4.

Table 4.4: Elasticity PDE: Excution Time of Numeric $\mathsf{C} = \mathsf{RAR}^T$ (seconds)
As the matrix size increases the number of colors increases more quickly with the LF coloring than the SL coloring. This causes the SL time to scale more effectively, although both colorings outperform the "No Coloring" option.

Matrix Size (Fine Grids)		No Coloring	Coloring AR^T			
A	R		LF	n_{color}	SL	n_{color}
$3,000 \times 3,000$	$156 \times 3,000$	.022	.0092	48	.0092	48
$24,000 \times 24,000$	$1,122 \times 24,000$	.24	.15	66	.15	60
$192,000 \times 192,000$	$8,586 \times 192,000$	2.23	1.68	84	1.33	66
$648,000 \times 648,000$	$27,924 \times 648,000$	7.86	6.07	90	4.75	69

Table 4.5: Elasticity PDE: Execution Time of Numeric $\mathsf{C} = \mathsf{GG}^T$ (seconds)
As the matrix G size grows, the number of colors from the LF coloring remains constant, allowing it to outperform the SL coloring. Again, both coloring choices are faster than the "No Coloring" option.

Matrix Size (Fine Grids)	No Coloring	Coloring			
		LF	n_{color}	SL	n_{color}
$1,000 \times 1,000$	.012	.0026	125	.0029	136
$8,000 \times 8,000$	.13	.038	125	.043	149
$64,000 \times 64,000$	1.22	.42	125	.51	158
$216,000 \times 216,000$	4.31	1.49	125	1.81	161

As the mesh size increases for matrices AR^T, the number of colors produced by the SL algorithm grows slower than that of LF: from 48 to 69 vs. 48 to 90. This results in improved performance using SL colorings as shown in Table 4.4. This stands in contrast to the results for the graph connection matrices GG^T show in Table 4.5 where the LF algorithm generates a more consistent number of colors as the mesh size grows, and outperforms the SL algorithm. These results suggest that no one coloring algorithm is ideal for all circumstances. Note that for all the cases presented, the coloring algorithms outperform the sparse inner product approach of "No Coloring".

5. Conclusions and Future Work. Earlier research has accelerated the computation of sparse Jacobians by using matrix coloring to evaluate several columns simultaneously. This work has been adapted to accelerate sparse matrix products computed with sparse inner products by applying a matrix coloring to instead compute a related compressed sparse-dense matrix product. We have proved that for the product $\mathsf{C} = \mathsf{AB}^T$, the matrix coloring of C is always a viable choice for compressing B^T, although slight modifications may be necessary if A has any zero columns.

Algorithms for both (2.1a) and (2.1b) were proposed, as well as considerations for practical implementation. Numerical results suggested that simulations using multigrid that computed sparse matrix products through inner products can be accelerated significantly through the use of coloring. Those results can be improved by fine tuning the data traffic during the decompression of the computed dense matrix to the desired sparse matrix. We also studied the effect of the choice of coloring algorithm on the efficiency of the compression and found that different algorithms perform better in different circumstances.

Future adaption of this coloring approach to parallel matrices will inherit much of the theory, but the added cost of data traffic may require some retooling or reorganization. Additionally, it would be valuable to study the use of this method on block matrices, a common structure for many applications. Beyond multigrid, our motivating application, we believe that applications from the discrete math community (e.g., breadth-first search) will benefit from our coloring algorithm, and we need to research this possibility.

The transition to a sparse-dense matrix product opens the door to numerous optimizations that are unavailable in the sparse-sparse setting. For instance, our current software computes four columns of the dense matrix each time the sparse matrix is loaded into memory, thereby reducing the total cost of accessing the sparse matrix. Another improvement we hope to consider is the interlacing of several columns of the dense matrix during the product to allow for more computation per sparse matrix cache miss. Decompressing the resulting product requires new techniques, so more work is required to take advantage of this and other previously unavailable opportunities.

We plan to incorporate the block dense to sparse decompression discussed in Section 3.1 into the PETSc library for faster computation of finite-difference Jacobians using coloring. Another interesting advance might be to develop a new coloring algorithm that incorporates the cost of decompression when deciding how to organize colors. A coloring algorithm that accelerates the decompression would benefit both the matrix multiplication setting and the evaluation of finite difference Jacobians.

REFERENCES

[1] M. F. Adams, H. H. Bayraktar, T. M. Keaveny, and P. Papadopoulos, *Ultrascalable implicit finite element analyses in solid mechanics with over a half a billion degrees of freedom*, in ACM/IEEE Proceedings of SC2004: High Performance Networking and Computing, 2004. Gordon Bell Award.

[2] S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, *PETSc users manual*, Tech. Report ANL-95/11 - Revision 3.4, Argonne National Laboratory, 2013.

[3] T. F. Coleman and J. J. More, *Estimation of sparse Jacobian matrices and graph coloring problems*, SIAM Journal on Numerical Analysis, 20 (1983), pp. 187–209.

[4] G. H. Golub and C. F. Van Loan, *Matrix Computations (4th ed.)*, Johns Hopkins University Press, Baltimore, MD, 2012.

[5] G. E. Hammond, P. C. Lichtner, and R. T. Mills, *Evaluating the performance of parallel subsurface simulators: An illustrative example with PFLOTRAN*, Water Resources Research, (2013).

[6] W. Klotz, *Graph coloring algorithms*, Mathematics Report, (2002), pp. 1–9.

[7] M. Kubale, *Graph Colorings*, Contemporary Mathematics (American Mathematical Society) v. 352, American Mathematical Society, 2004.

[8] P. Lichtner et al., *PFLOTRAN project*. `http://ees.lanl.gov/pflotran/`.

[9] D. W. Matula and L. L. Beck, *Smallest-last ordering and clustering and graph coloring algorithms*, J. ACM, 30 (1983), pp. 417–427.

[10] Y. Saad, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, Boston, MA, 1996.

[11] B. Smith and H. Zhang, *Sparse triangular solves for ILU revisited: Data layout crucial to better performance*, International J. High Performance Computing Applications, 25 (2011), pp. 386–391.

[12] K. Stüben, *A review of algebraic multigrid*, J. Comput. Appl. Math., 128 (2001), pp. 281–309.

[13] U. Trottenberg, C. W. Oosterlee, and A. Schuller, *Multigrid*, Elsevier Science, 2000.